Final Project: Large-scale Collaborative Ranking from Pairwise Comparisons

Liwei Wu

December 4, 2016

1 Overview

The problem we are considering is collaborative ranking setting: a pool of users each provides a small of users provides a small number of pairwise preferences between $\bar{d_2}$ possible items and we want to predict each user's preferences for items that they have not yet seen. It is different from RankSVM: aim for personaliztion despite insufficient information for any particular user in isolation. The approach is fitting a rank r score matrix to the pairwise data by solving the proposed objective function. There is existing algorithm using stochastic dual coordinate descent algorithm to solve this problem, but it has time complexity of $O(|\Omega|r) \approx O(d_1\bar{d_2}^2 r)$, where d_1 is the number of users and r is the rank of the score matrix. In this project, we came up with a new algorithm, which has better time complexity in theory than the existing algorithm AltSVM in the literature. We tried an initial implemenatation of the alogirthm in Julia and evaluated its performance.

2 Mathematical Formulation

The recommendation system problem we are interested in is that given a set of items, a set of users, and non-numerical pairwise comparison data, find the underlying preference ordering of the users. The data is of the form "user i prefers item j over item k", i.e. different ordered user-item-item triples (i, j, k). We denote the number of users by d_1, and the number of items by d_2. We are given a set of triples $\Omega \subset [d_1] \times [d_2] \times [d_2]$, where the preference of user i between items j and k is observed if $(i, j, k) \in \Omega$. The observed comparison is given by $\{Y_{ijk} \in \{1, -1\} : (i, j, k) \in \Omega\}$, where $Y_{ijk} = 1$ is user i prefers item j over item k, and $Y_{ijk} = -1$ otherwise.

We predict rankings for multiple users by estimating a score matrix $X \in \mathbb{R}^{d_1 \times d_2}$ such that $X_{ij} > X_{ik}$ means that user i prefers item j over item k. Then sorting order for each row provides the predicted ranking for the corresponding user.

We propose X is low-rank, due to the intuition that user bases their preferences on a small set of features that are common among all the items. Then we can formulate the empirical risk minimization (ERM) framework as:

$$\min_X \sum_{(i,j,k)\in\Omega} L(Y_{ijk}(X_{ij} - X_{ik})) \tag{1}$$

$$\text{subject to} \quad \text{rank}(X) \leq r \tag{2}$$

, where $\mathcal{L}(.)$ is a montonically non-increasing loss function such as L_1 and L_2 hinge loss and logistic loss etc.

Because solving the equation above is NP-hard, for the large-scale datasets we can solve the non-convex objective function:

$$\min_{U,V} \sum_{i=1}^{d_1} \sum_{j,k\in\bar{d_2}(i)} \mathcal{L}(Y_{ijk} * u_i^T(v_j - v_k)) + \frac{\lambda}{2}(||U||_F^2 + ||V||_F^2)$$

, where $U \in \mathbb{R}^{d_1 \times r}$ and $V \in \mathbb{R}^{r \times d_2}$ so that X can be recovered as $U \times V$ in the end.

3 Optimization Algorithm

We propose that the primal method be used to solve the objective algorithm: fix V and optimize with respect to U and then fix U and optimize with respect to V. Updating V and U alternatively finds the optimal V and U in the end.

3.1 Fix U, Update V

While U is fixed:

$$V = \underset{V\in\mathbb{R}^{r\times d_2}}{\operatorname{argmin}}\{\frac{\lambda}{2}||V||_F^2 + \sum_{i=1}^{d_1}\sum_{(j,k)\in\bar{d}_2(i)}\mathcal{L}(u_i^T(v_j - v_k))\} \tag{3}$$

The gradient of V is

$$\nabla f(V) = \sum_{i=1}^{d_1}\sum_{(j,k)\in\bar{d}_2(i)}\mathcal{L}'(u_i^T(v_j - v_k))(u_iE_j - u_iE_k) + \lambda V \tag{4}$$

, where $\nabla f(V) \in \mathbb{R}^{r\times d_2}$ and E_j, E_k specficies which column in $\nabla f(V)$ to add.

So the pseudocodes for obtaining the gradient g are

1. initialize $g = \lambda V$
2. for $i = 1, 2, ..., d_1$
   - precompute $u_i^T v_j$ for all $j \in \bar{d}_2(i)$
   - for $(j,k) \in \bar{d}_2(i)$
     - compute $m = u_i^T v_j - u_i^T v_k$
     - if $m < 1$: $s_{jk} = 2*(m-1); t_j{+}= s_{jk}; t_k{-}= s_{jk}$
     - else: continue
   - for $j = 1, ..., \bar{d}_2$: $g_j{+}= t_j * u_i$

This step has time complexity $O(d_1 * (\bar{d}_2 r + \bar{d}_2{}^2))$.

In order to solve δ from $H\delta = g$ using linear conjugate gradient descent algorithm, we also need to express the Hessian matrix H explicitly and obtain Ha quickly for any vector a. Since one may realize for this case $g \in \mathbb{R}^{r\times d_2}$, it is hard to form the Hessian matrix. We can first vectorize the graident g and obtain $g \in \mathbb{R}^{d_2 r}$ and then obtain $H \in \mathbb{R}^{d_2 r\times d_2 r}$.

After some derivation,

$$\nabla^2_{p,p} f(V) = \sum_{i:p\in\bar{d}_2(i)}\sum_{k\in\bar{d}_2(i),k\neq p}\mathcal{L}''(y_{ipk}u_i^T(v_p - v_k))u_iu_i^T + \lambda I_{r\times r} \tag{5}$$

where $\nabla^2_{p,p} f(V) \in \mathbb{R}^{r\times r}$, the subscript p denotes the pth column of V: v_p.

And

$$\nabla^2_{p,q} f(V) = \sum_{i:(p,q)\in\bar{d}_2(i)}\mathcal{L}''(y_{ipk}u_i^T(v_p - v_k))u_iu_i^T(-1) \tag{6}$$

Therefore, the product of $H \in \mathbb{R}^{d_2 r\times d_2 r}$ and $a \in \mathbb{R}^{d_2 r}$ is

$$(Ha)_p = \lambda a + \sum_i \sum_{p\in\bar{d}_2(i)} E_p u_i \sum_{q\in\bar{d}_2(i),q\neq p}\mathcal{L}''(y_{ipq}u_i^T(v_p - v_q))(u_i^T a_p - u_i^T a_q) \tag{7}$$

So the pseudocodes for obtaining the product of H and a are

1. initialize $(Ha)_p = \lambda a_p$ for all p
2. for $i = 1, 2, ..., d_1$
   - precompute $u_i^T a_q$ for all $q \in \bar{d}_2(i)$
   - for $p \in \bar{d}_2(i)$
     - $c_p = 0$
     - for $q \in \bar{d}_2(i)$ and $q \neq p$
       * compute $m = y_{ipq}(u_i^T v_p - u_i^T v_q)$
       * if $m < 1$: $s_{pq} = 2$
       * else: $s_{pq} = 0$
       * $c_p = c_p + s_{pq}(b_p - b_q)$
     - $(Ha)_p = (Ha)_p + u_i c_p$

This step also has time complexity $O(d_1 * (\bar{d}_2 r + \bar{d}_2{}^2))$.

For given U, one can obtain corresponding g and Ha for any given vector a in $O(d_1 * (\bar{d}_2 r + \bar{d}_2{}^2))$, then we can use linear conjugate gradient descent algorithm to solve δ from $H\delta = g$.

1. initialize $\delta_0 = 0$
2. $r_0 = H\delta_0 - g$, $p_0 = -r_0$
3. for $k = 0, 1, ..., n-1$
   - $\alpha_k = -r_k^T p_k / p_k^T H p_k$
   - $\delta_{k+1} = \delta_k + \alpha_k p_k$
   - $r_{k+1} = r_k + \alpha_k H p_k$
   - $\beta_{k+1} = (r_{k+1} H p_k) / p_k^T H p_k$
   - $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$

After obtaining δ_n as δ, then we can update V using truncated Newtown method:

$$V = V - t\delta \tag{8}$$

, where t is the step size and can be chosen using line search.

3.2 Fix V, Update U

Similar to the previous section, we also use the truncated Newton method together with linear conjugate gradient descent algorithm, but we obtain g and Ha in a different way and instead of optimizing with respect to U, we can simply optimize with respect to u_i while V is fixed:

$$u_i = \underset{u \in \mathbb{R}^r}{\text{argmin}} \frac{\lambda}{2} ||u||_F^2 + \sum_{(j,k) \in \bar{d}_2(i)} \mathcal{L}(u^T(v_j - v_k)) \tag{9}$$

There are two different ways to obtain g and Hs for any vector s:

For the **first** approach:

1. First we form diagonal matrix $D \in \mathbb{R}^{\bar{d}_2{}^2 \times \bar{d}_2{}^2}$ with diagonal element defined as

$$D_{ii} = \mathbb{1}(u^T v_j - u^T v_k < 1) \tag{10}$$

2. obtain g:
$$g = \lambda u + 2\bar{V} A^T D(A\bar{V}^T u - \mathbb{1}) \tag{11}$$

3. obtain the product of H and any vector s using
$$Hs = \lambda s + 2\bar{V} A^T D A \bar{V}^T s \tag{12}$$

This approach has time complexity $O(d_1 * (\bar{d_2} r + \bar{d_2}^2))$.

For the **second** approach, motivated by [Chapelle and Keerthi(2010)]:

$$u_i = \underset{u}{\operatorname{argmin}} \frac{\lambda}{2} ||u||_F^2 + \sum_{l=2}^{R} \sum_{\substack{(j,k)\in\bar{d_2}(i)\\(i,j)\to l\\(i,k)\to < l}} \mathcal{L}(u^T(v_j - v_k)) \tag{13}$$

, where R represents the number of preference levels.

And the pseudocodes for obtaining g and Hs are

1. for each $j \in \bar{d_2}(i)$
   - compute $y_j = u^T v_j$
   - compute $y'_j = y_j - \frac{1}{2}$ and $y''_j = y_j + \frac{1}{2}$
2. sort all y'_j and y''_j
3. initialize $g = \lambda u, Hs = \lambda s$
4. for $l = 2, 3, ..., R$:
   - find y_j and y_k such that (i,j) is in preference level l (i.e. $(i,j) \to l$), and (i,k) in a preference level lower than l (i.e. $(i,k) \to < l$). Denote the higher relveance class that y_j belongs to as A and lower relevance class that y_k belongs to as B.
   - define subclass B_j and A_k as
   $$B_j = \{i \in B : \tilde{y}_i > \tilde{y}_j\}, j \in A \tag{14}$$
   $$A_k = \{i \in A : \tilde{y}_i < \tilde{y}_k\}, k \in B \tag{15}$$
   - obtain α_i, β_i for $i = 1, ..., n_l$ using following formula
   $$\alpha_j = |B_j|, \alpha_k = |A_k|, \beta_j = \sum_{i\in B_j} \tilde{y}_i, \beta_k = \sum_{i \in A_k} \tilde{y}_i \tag{16}$$
   , where n_l denotes the number of all possible preference levels for j or k, i.e. $j, k \to \leq l$.
   - update g:
   $$g = g + \tilde{V} \frac{\partial l}{\partial y} \tag{17}$$
   , where $\tilde{V} \in \mathbb{R}^{r\times n_l}$, $\frac{\partial l}{\partial y} \in \mathbb{R}^{n_l}$ and
   $$\frac{\partial l}{\partial y_i} = 2(\alpha_i \tilde{y}_i - \beta_i), \forall i = 1, ..., n_l \tag{18}$$
   - update Hs:
   $$\begin{aligned} Hs &= Hs + 2\tilde{V} z \\ z_i &= \alpha_i \delta_i - \gamma_i \end{aligned} \tag{19}$$

, where $\tilde{V} \in \mathbb{R}^{r \times n_l}, \delta \in \mathbb{R}^{n_l}$ and

$$\begin{aligned} \delta &= \tilde{V}^T s \\ \gamma_j &= \sum_{k \in A_j} \delta_k, j \in A \\ \gamma_k &= \sum_{j \in A_k} \delta_j, k \in B \end{aligned} \tag{20}$$

This approach has time complexity $O(d_1 * (\bar{d}_2 r + \bar{d}_2 \log \bar{d}_2))$.

Using either approach out of two, together with linear conjugate gradient and truncated newton method allows us to update U while V fixed in $O(d_1 * (\bar{d}_2 r + \bar{d}_2 \log \bar{d}_2))$ optimally or in $O(d_1 * (\bar{d}_2 r + \bar{d}_2{}^2))$.

4 Data Set and Implementation

The data set we are testing against is MovieLens1m data. It contains 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,400 MovieLens users who joined MovieLens in 2000. The data is of the form: User ID::Movie ID::Rating. We implemented the algorithm in Julia. Due to time constraint, we only had time to implment the first approach for the subsection "Fix V, Update U". One can easily follow codes attached in the Code section with the pseudocodes listed in the previous section.

5 Results

Running the codes on the server in the Statistics Department called "Poisson", takes more than 30 mins to converge. According to the paper [Sanghavi and Dhillon(2015)], the existing algorithm takes roughly 16 mins to converge. So despite the fact that proposed algorithm has better time complexity in theory, my initial implementation does not run as fast as the existing one.

6 Conclusion and Future Work

Although my initial implementation does not run as fast the existing one, but there are possible reasons for it and there exists room for improvement. First, I did not implement the complicated second approach in the subsection "Fix V, Update U", which has better time complexity and is expected to run faster. Second, the codes are written in Julia while the existing one is written in C++. Last, the data I use is not preprocessed into the the form: "user i prefers item j over item k", i.e. different ordered user-item-item triples (i, j, k), while the existing algorithm assumes that data is already in that particular form so it saves time in the preprocessing process.

References

[Chapelle and Keerthi(2010)] O. Chapelle and S. S. Keerthi. Efficient algorithms for ranking with svms. *Information Retrieval*, 13(3):201–215, 2010.

[Sanghavi and Dhillon(2015)] S. Sanghavi and I. S. Dhillon. Preference completion: Large-scale collaborative ranking from pairwise comparisons. 2015.

7 Codes

```
# python
import numpy as np
from numpy import genfromtxt
import scipy
import random
from scipy.sparse import csr_matrix, csc_matrix, coo_matrix, dok_matrix
from scipy import sparse, io
data = genfromtxt('ratings.dat', delimiter='::')
data = np.delete(data, 3, 1)
data = np.delete(data, 0, 0)
x = data[:,0]
y = data[:,1]
n = int(max(x))
m = int(max(y))
x = x - 1
y = y - 1
v = data[:,2]

import csv
f = open('MovieLens1m.csv', 'w')
writer = csv.writer(f)
for i in range(len(v)):
        writer.writerow( (int(x[i]), int(y[i]), int(v[i]) ))

f.close()

# julia

# Fix U, update V
function helper(m, t, i, j, k, J, K)
        mask = m[J, i] - m[K, i]
        if mask >= 1.0
                return t
        else
                s_jk = 2.0 * (mask - 1.0)
                t[j] += s_jk
                t[k] -= s_jk
        end
        return t
end

function obtain_g(U, V, X, d1, d2, lambda, rows, vals)
        g = lambda * V;
        m = spzeros(d2,d1);
        #aa=0
        for i = 1:d1
                tmp = nzrange(X, i)
                d2_bar = rows[tmp];
                ui = U[:, i]
                for j in d2_bar
                        m[j,i] = dot(ui,V[:,j])
                end

                vals_d2_bar = vals[tmp];
                len = size(d2_bar)[1];
                t = spzeros(1, len);
                for j in 1:(len - 1)
                        J = d2_bar[j];
                        for k in (j + 1):len
                                K = d2_bar[k];
                                #aa+=1
                                if vals_d2_bar[j] > vals_d2_bar[k]
                                        t = helper(m, t, i, j, k, J, K)
                                elseif vals_d2_bar[k] > vals_d2_bar[j]
                                        t = helper(m, t, i, k, j, K, J)
                                end
                        end
```

```
                end

                for j in 1:len
                        J = d2_bar[j]
                        g[:,J] += ui * t[j]
                end
        end
        #println("aa:$(aa)")
        return g, m
end

function compute_Ha(a, m, U, X, r, d1, d2, lambda, rows, vals)
        Ha = lambda * a
        for i in 1:d1
                tmp = nzrange(X, i)
                d2_bar = rows[tmp]
                b = spzeros(1,d2)
                ui = U[:,i]
                for q in d2_bar
                        a_q = a[(q-1)*r+1:q*r]
                        b[1,q] = dot(ui, a_q)
                end

                vals_d2_bar = vals[tmp]
                len = size(d2_bar)[1]
                for j in 1:(len - 1)
                        p = d2_bar[j];
                        c_p = 0.0
                        for k in (j + 1):len
                                q = d2_bar[k]
                                if vals_d2_bar[j] == vals_d2_bar[k]
                                        continue
                                elseif vals_d2_bar[j] > vals_d2_bar[k]
                                        y_ipq = 1.0
                                elseif vals_d2_bar[k] > vals_d2_bar[j]
                                        y_ipq = -1.0
                                end
                                mask = y_ipq * (m[p, i] - m[q, i])
                                if mask >= 1.0
                                        continue
                                else
                                        s_pq = 2.0
                                        c_p += s_pq * (b[1,p] - b[1,q])
                                end
                        end
                        Ha[(p - 1) * r + 1 : p * r] += ui * c_p
                end
        end
        return Ha
end

function solve_delta(g, m, U, X, r, d1, d2, lambda, rows, vals)
        # use linear conjugate grad descent
        delta = zeros(size(g)[1])
        rr = -g
        p = -rr
        err = norm(rr) * 10.0^-3
        for k in 1:10
                #println(k)
                Hp = compute_Ha(p, m, U, X, r, d1, d2, lambda, rows, vals)
                alpha = -dot(rr, p) / dot(p, Hp)
                delta += alpha * p
                rr += alpha * Hp
                if norm(rr) < err
                        break
                end
                #println(norm(rr))
                b = dot(rr, Hp) / dot(p, Hp)
                p = -rr + b * p
        end
```

```
        return delta
end


function objective(m, U, V, X, d1, lambda, rows, vals)
        res = lambda / 2 * (vecnorm(U) ^ 2 +vecnorm(V) ^ 2)
        for i in 1:d1
                tmp = nzrange(X, i)
                d2_bar = rows[tmp];
                vals_d2_bar = vals[tmp];
                len = size(d2_bar)[1];
                for j in 1:(len - 1)
                        p = d2_bar[j];
                        for k in (j + 1):len
                                q = d2_bar[k]
                                if vals_d2_bar[j] == vals_d2_bar[k]
                                        continue
                                elseif vals_d2_bar[j] > vals_d2_bar[k]
                                        y_ipq = 1.0
                                elseif vals_d2_bar[k] > vals_d2_bar[j]
                                        y_ipq = -1.0
                                end
                                mask = y_ipq * (m[p, i] - m[q, i])
                                if mask >= 1.0
                                        continue
                                else
                                        res += (1.0 - mask) ^ 2
                                end
                        end
                end
        end
        return res
end


function update_V(U, V, X, r, d1, d2, lambda, rows, vals, stepsize)
        g,m = obtain_g(U, V, X, d1, d2, lambda, rows, vals)
        g = vec(g)
        delta = solve_delta(g, m, U, X, r, d1, d2, lambda, rows, vals)
        delta = reshape(delta, size(V))
        obj = objective(m, U, V, X, d1, lambda, rows, vals)
        println("objective function value:", obj)
        V -= stepsize * delta
        return V
end


# Fix V, update U
function helper2(i, ui, V, X, r, d2, rows, vals)
        tmp = nzrange(X, i)
        d2_bar = rows[tmp];
        m = spzeros(1, d2)
        # need to get new m for updated
        for j in d2_bar
                m[1,j] = dot(ui,V[:,j])
        end

        vals_d2_bar = vals[tmp];
        len = size(d2_bar)[1];
        num = round(Int64, len*(len-1)/2)
        D = zeros(num)
        A = spzeros(len, num)
        V_bar = zeros(r, len)
        c = 0
        for j in 1:len
                p = d2_bar[j];
                V_bar[:,j] = V[:,p]
                for k in (j + 1):len
                        q = d2_bar[k]
                        if vals_d2_bar[j] == vals_d2_bar[k]
```

```
                                continue
                        elseif vals_d2_bar[j] > vals_d2_bar[k]
                                y_ipq = 1.0
                                c += 1
                                A[j, c] = 1.0; A[k, c] = -1.0
                        elseif vals_d2_bar[k] > vals_d2_bar[j]
                                y_ipq = -1.0
                                c += 1
                                A[j, c] = -1.0; A[k, c] = 1.0
                        end
                        mask = y_ipq * (m[1, p] - m[1, q])
                        #println(mask)
                        if mask >= 1.0
                                continue
                        else
                                D[c] = 1.0
                        end
                end
        end
        D = D[1:c]; A = A[:,1:c]
        D = spdiagm(D)
        return A, D, V_bar, m
end


function obtain_g_u(A, D, V_bar, ui, lambda)
        tmp = A' * (V_bar' * ui)
        tmp -= ones(size(A)[2])
        tmp = D * tmp
        tmp = A * tmp
        tmp = 2 * V_bar * tmp
        tmp += lambda * ui
        return tmp
end

function obtain_Hs(s, A, D, V_bar, lambda)
        tmp = A' * (V_bar' * s)
        tmp = D * tmp
        tmp = A * tmp
        tmp = 2 * V_bar * tmp
        tmp += lambda * s
        return tmp
end

function solve_delta_u(g, A, D, V_bar, lambda)
        # use linear conjugate grad descent
        delta = zeros(size(g)[1])
        rr = -g
        p = -rr
        err = norm(rr) * 10.0^-3
        for k in 1:10
                #println(k)
                Hp = obtain_Hs(p, A, D, V_bar, lambda)
                alpha = -dot(rr, p) / dot(p, Hp)
                delta += alpha * p
                rr += alpha * Hp
                if norm(rr) < err
                        break
                end
                #println(norm(rr))
                b = dot(rr, Hp) / dot(p, Hp)
                p = -rr + b * p
        end
        return delta
end

function objective_u(i, m, X, lambda, rows, vals)
        res = lambda / 2 * (vecnorm(ui) ^ 2)
        tmp = nzrange(X, i)
        d2_bar = rows[tmp];
```

```
        vals_d2_bar = vals[tmp];
        len = size(d2_bar)[1];
        for j in 1:(len - 1)
                p = d2_bar[j];
                for k in (j + 1):len
                        q = d2_bar[k]
                        if vals_d2_bar[j] == vals_d2_bar[k]
                                continue
                        elseif vals_d2_bar[j] > vals_d2_bar[k]
                                y_ipq = 1.0
                        elseif vals_d2_bar[k] > vals_d2_bar[j]
                                y_ipq = -1.0
                        end
                        mask = y_ipq * (m[1, p] - m[1, q])
                        if mask >= 1.0
                                continue
                        else
                                res += (1.0 - mask) ^ 2
                        end
                end
        end
        return res
end

function update_u(i, ui, V, X, r, d2, lambda, rows, vals, stepsize)
        A, D, V_bar, m = helper2(i, ui, V, X, r, d2, rows, vals)
        g = obtain_g_u(A, D, V_bar, ui)
        delta = solve_delta_u(g, A, D, V_bar, lambda)
        obj = objective_u(i, m, X, lambda, rows, vals)
        println("objective function value:", obj)
        ui -= stepsize * delta

        return ui, obj
end

function update_U(U, V, X, r, d1, d2, lambda, rows, vals, stepsize)
        for i in 1:d1
                println("Outer iteration:", i)
                ui = U[:, i]
                prev = 0
                for k in 1:100
                        println(k)
                        ui, obj = update_u(i, ui, V, X, r, d2, lambda, rows, vals, stepsize)
                        if k == 1
                                prev = obj
                        else
                                if (prev - obj) / prev < 10.0 ^ -3
                                        break
                                end
                                prev = obj
                        end
                        println(prev)
                end
                U[:, i] = ui
        end
        return U
end

function main()
        X = readdlm("MovieLens1m.csv", ',' , Int64);
        x = vec(X[:,1]) + 1;
        y = vec(X[:,2]) + 1;
        v = vec(X[:,3]);
        n = 6040; m = 3952;
        X = sparse(x, y, v, n, m);
        # julia column major
        X = X'
        rows = rowvals(X)
        vals = nonzeros(X)
        d2, d1 = size(X)
```

```
        r = 20; lambda = 5000;
        # initialize U, V
        U = 0.1*randn(r, d1);
        V = 0.1*randn(r, d2);
        stepsize = 1
        for iter in 1:10
                V = update_V(U, V, X, r, d1, d2, lambda, rows, vals, stepsize)

                U = update_U(U, V, X, r, d1, d2, lambda, rows, vals, stepsize)
        end
end
```